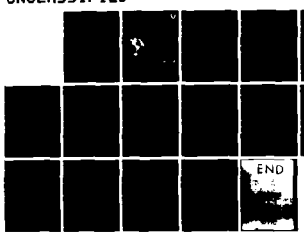
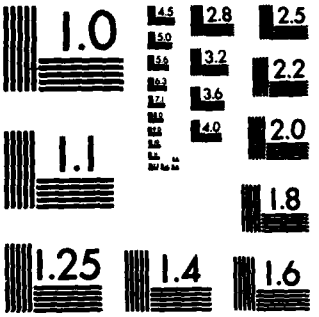
AD-A144 232 EMBEDDED SYSTEM DESIGN WITH ADA AS THE SYSTEM DESIGN 1/1
LANGUAGE(U) ARMY COMMUNICATIONS RESEARCH AND
DEVELOPMENT COMMAND FORT MONMOUTH NJ T J WHEELER 1984
UNCLASSIFIED F/G 9/2 NL





EMBEDDED SYSTEM DESIGN WITH ADA AS THE SYSTEM DESIGN LANGUAGE

THOMAS J. WHEELER

SOFTWARE TECHNOLOGY DIVISION

CORADCOM

U.S. ARMY COMMUNICATIONS RESEARCH AND DEVELOPMENT COMMAND

84 08 09 098

ABSTRACT

Recent research in the software engineering field has produced a number of techniques or rationals for structuring the understanding of systems. Many of these techniques are applicable to the design of embedded computer systems and produce designs whose structures are easily expressible in the Ada language. The Ada language has a structure which allows the design of systems to be expressed independently of its implementation and thus can be a good system design language for use with these techniques.

This paper describes the software design problem in the development of embedded computer systems and shows how the Ada language can be used as a system design language as well as a system implementation language to alleviate these problems. The essential point of this paper is that using Ada as a system design language encourages the designers to use the recently developed techniques and theory to develop better structures for their systems and then implement the systems in the same language thus preserving that structure in the product.

CONTENTS

I. Introduction.

In response to the explosive growth in the cost of development and maintenance of software systems, there have been a large number of theories and techniques developed in the area of software design and development. Some of these are structured programming, top-down design and implementation, structured analysis and design(21), stepwise refinement(23), information hiding(18) and programming teams and walkthroughs(24). The central aim of all of these is to provide intellectual control of a design by a systematic decomposition and abstraction of the problem into component modules and composition of these modules into the system.

While most of these techniques have produced impressive results, with measured gains of 4-6 times increase in productivity(2) not being uncommon, the use of these techniques in the embedded systems area has been limited. The reasons for this are varied. Some are technical such as lack of a suitable high level language and the techniques and compilers to go with them. However, other reasons are psychological as for instance, that the time to investigate design techniques and to learn to use a new language is viewed as not affordable during these normally time constrained developments.

The technical barriers to the use of modern software engineering theories and techniques are being overcome with the introduction of a language and techniques specifically designed for embedded computer applications. This paper addresses the effects of the Department of Defense's Ada language(1) on the design process for embedded systems. One of the reasons that the Ada language is so important to the design process is that the Ada language is structured to allow it to be used as a system design language as well as a programming language. A system design language (SDL) is a formal means of documenting the structure of the design of a system without the necessity of providing or refering to an implementation of the system. Ada provides this means by separating the specifications of the components from their implementations and by allowing interconnection of components only by those means documented in the specifications of the components.

One of the main themes of this paper is that the constraints on the system structure imposed by the use of the Ada language as the means for documenting the system's design not only cause the system's design and implementation to be easier but also cause the resulting system to be more maintainable. Additionally, the use of Ada as both the design and implementation language causes the documentation of the system to be more controlable since the major part of the documentation, even at the design level, is the system (ie. the program) itself.

One of the main criteria used in the design of the Ada language was that the language should aid in the design of reliable systems(5,22). This criteria led to the incorporation of modularization by packaging of named entities as the main basis(20) for structuring of software systems. In addition Ada provides a distinct separation of the specification of the visible named entities of the module from the implementation of the module. This allows the structure, or the architecture, of the system to be documented as the interconnection of the interfaces of the modules without reference to the implementations of the modules. The use of Ada as a system design language is a result of this ability to document the structure of a system using only the specifications of packages and their interconnection.

II. Modularization.

The worlds of mechanical design and electronic system design have long used the concept of modularization and have well developed methods of documenting designs in terms of their component modules, ie. blueprints and schematic drawings respectively. Ada provides a means of documenting software designs and communicating those design to others which, when supplemented with its equivalent graphic drawing, is the equal of the more mature documentation methods mentioned above. It is the equal because the basis is the same. The basis is that a design is represented as an interconnection of the interface characteristicts of components. This interconnection is a model of a well structured understanding of the system(21). It is the interface characteristics which actually define the components which are used in the design because the interface characteristics are all that the user of the component needs to use the component and all that the designer of the component needs to build the component(18).

The view of modularization embodied in Ada has evolved slowly over the past decade. The main reason for this slowness is that of the two means of modularization, decomposition and abstraction, decomposition was viewed as the method of modularization while abstraction was viewed as a mental tool rather than as a language supportable mechanism. In view of the way a programming language influences the way that people think about systems and vice versa, this was both the result and the cause of the structure of earlier high order languages such as Fortran, Cobol and Algol. In systems built in these languages, the interconnection of the major sub-tasks of the system was viewed as the responsibility of the operating system functions such as linkage editing and the system generation process. The interconnection of the smaller parts of the systems built in these languages was through the use of global or common data accessed by the subprograms from which the systems were constructed.

Modularization by abstraction had its roots in the virtual machine concept(6) and has been influenced by most of the major advances made by software engineering research, eg. the data typing mechanism of Pascal(14), information hiding(18,19), abstract data types(7,8) and module interconnection languages(4). The consensus developed in the research results is that a software system can and should be designed and constructed as an interconnected network of software objects of abstract data types. Abstract data types are constructed out of a set of values, which may be a complex composite of simpler values, and a set of operations which is applicable to the values, with no other operations allowed. Each of the objects of these types is to represent (encapsulate)a particular logical entity such as a design decision(18) or a related set of properties of a logical item(7).

A graphical representation of a system modularized in this way is shown in FIG 1a. This system prints reports from local files or, if the report is not available in the local files, the report manager requests it from remote files and prints it when it has been copied to the local files. The explanation of this diagramming method is in FIG 1b which is a diagram of a single generic module where the abstract type or object is indicated by the named box and the resources, eg. types, functions etc., which are provided by the module and those which are required by the module are indicated by the outgoing arrow and the incomming arrow respectively.

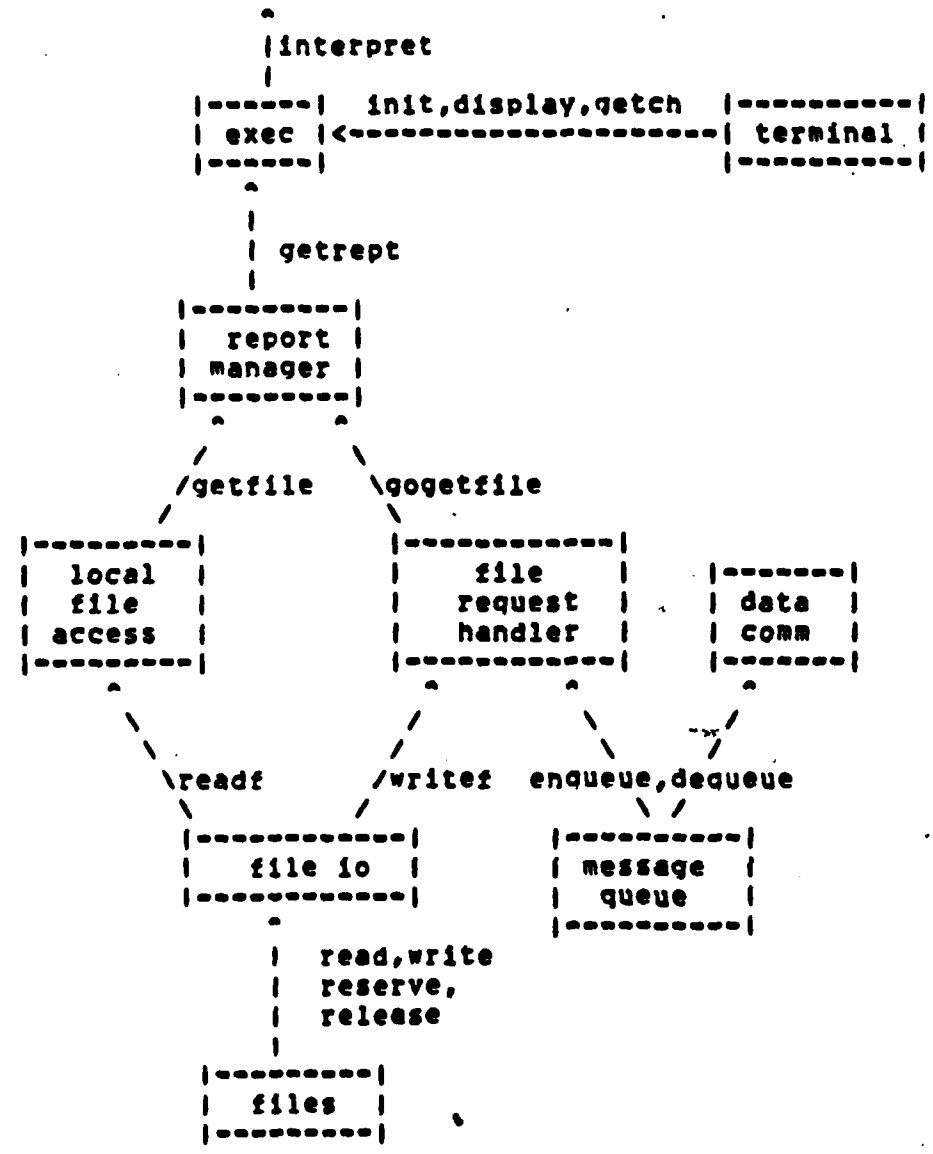


FIG 1a

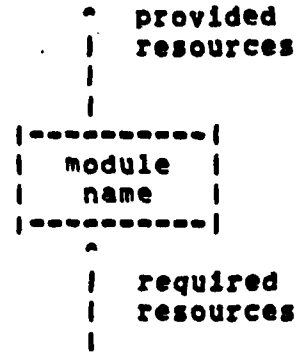


FIG 1b

The Ada language encourages this style of design by having packaging of data and procedures as its large scale structuring mechanism. In the system diagrammed in FIG 1a, each of the modules becomes a package in Ada. For example, the files module in FIG 1a which provides access to the local file system is specified in Ada as:

```
package FILES is
  type FILE is STRING(32);
  type LINE is STRING(80);
  procedure READ(F:in FILE;L:out LINE);
  procedure WRITE(F:in FILE;L:in LINE);
  procedure RESERVE(F:in FILE);
  procedure RELEASE(F:in FILE);
end FILES;
```

Since a programming language influences the way that people think about systems, the use of Ada over a period of time leads the designers to the use of abstract data types as a natural way to visualize system designs.

III. The Ada Language.

The Ada language(1) was designed with three specific goals: 1.reliability and maintainability, 2.recognition of programming as a human activity and 3.efficiency. The first two of these criteria drove the structure of the language and its intended use while the last filtered the possible inefficient structures from the language. The following is a brief overview of the important features of the Ada language. The Ada language reference manual(1) should be consulted for a more thorough understanding of the language.

Program units.

Ada is designed to encourage modularization and the accompanying ability to factor and compose a system from separately built parts. In Ada, a program is composed from program units, which are subprograms, packages (which define collections of entities,ie. named items), or tasks (which define concurrent or parallel computations). Each of these program units is made up of two parts: a specification, which contains those entities that are visible to other program units thus defining the external characteristics of the unit, and a body, which contains the implementation of these entities and is not visible to other units. Units and their parts are separately compilable.

This separation of the specification and the implementation parts of modules along with the ability to separate compile these parts allows and encourages both the construction of systems from separately built parts and the construction and use of libraries of generally usable component modules.

Program unit specifications.

The program units from which Ada programs are consrtucted are: subprograms, packages and tasks. Program unit specifications are named declarations which provide the types, objects and operations which can be used by other program units.

Subprograms are the basic unit for expressing algorithms and provide the means for naming definable actions. The two kinds of subprograms are procedures and functions. A procedure provides the series of actions, defined in its body, whenever it is invoked. It may have parameters to pass information into itself or back to its invoker. A function is a named activity, or operation, which computes a value. It is similar to a procedure but returns a value as the result of its being invoked in an expression.

Packages are the units for encapsulating collections of logically related data. Packages define sets of related types, data, operations or subprograms. Only those entities which are defined in the visible or specification part of the package are allowed to be used by other units. The implementation of the visible entities is hidden in the body of the package.

Tasks are the program units for defining operations or procedures which execute in parallel with other tasks. Tasks define entries which are the synchronized operations provided for use by other tasks. Multiple instances of tasks or dynamic tasks can be declared as objects of task types.The language allows tasks to be implemented on multiprocessors, multicomputers or to be interleaved on a single processor.

Program unit bodies.

Program unit bodies consist of a declaritive part, which declares the entities which can be used in the unit, and a sequence of statements, which defines the implementation of that program unit's actions. These named entities which can be used by the sequence of statements include types, objects, exceptions and other nested program units.

The sequence of statements describes the unit's actions. The statements can include assignment of values to variables, procedure calls and structuring constructs. The structuring constructs include "if" and "case" statements for selection, "while" and "for" loops for iteration and blocks for temporary declarations and actions.

Tasks are constructed with the statements described above supplemented with real time and syncronization statements. These include the "delay" statement, the "entry" declaration for providing services to other tasks, and the "select" and "accept" statements for controling the rendezvous which syncronize tasks. In a rendezvous, either the requester or the provider of an action arrives at the rendezvous point before the other. The one which arrives first waits for the other. When the other arives, the rendezvous takes place, the action is performed and they both proceed with their next statements.

Exceptional conditions, which make it impossible to continue with the normal execution of the program, are handled by a sequence of statements enclosed in an exception handler placed at the end of the program unit. When an exception handler is invoked, it replaces the remainder of the unit where the exception occurred. Exceptions can be raised explicitly in the program.

Types

Every object and value in the program has a type. A type consists of a set of values and a set of operations applicable to the values. Types are divided into four classes: scalar, composite, access and private types.

Scalar types are both the numeric types(integer, fixed point and floating point) and enumeration types which allow programmers to define ordered sets of distinct enumeration literals to be used as values in the program. Composite types provide the means of defining structured objects formed from related components. Two kinds of composite types are arrays, which have indexed components of the same type, and records, which have named components of possibly different types.

Access types are used to construct dynamic data structures by defining a mechanism for accessing unnamed objects which are created by allocators. Both the contents of the objects and the access values to the objects may be changed by the program.

Private types are defined in packages and only their names are made visible in the specification part of the package. All operations on values of private type variables are defined in the package and both the structure of the data used to define the type and the algoritnms which implement the operations are hidden in the body of the package.

Other features

Ada provides a number of other facilities to provide the program designer with complete control of the computer when necessary. These include representation of entities, control of interrupts and machine code insertion. Input-output is provided as a library feature rather than being built into the language. The language provides for generic program units to encapsulate sets of algorithms applicable to various types.

IV. System Architecture Design.

The quality of a system is highly dependent on the language in which the system is designed and the language in which the system is built. The criteria in both cases is similar, the ease with which the requirements or problem structure can be modeled in the design, in the first case and the ease with which the design can be modeled in the implementation , in the second case. An ideal language would allow both the design and the implementation to have a structure which is an accurate recording of the solution to the problem.

Almost all of the current high level languages discourage this sort of mapping of solution into implementation. These languages sole concern is in the expression of data and algorithms for a single process. This is fine for small programs but inadequate for large systems. These languages have been refered to as languages for programming in the small(4). A system design language SDL provides the means for describing the interconnection information which is the essence of system structuring and is analogously refered to as a language for programming in the large(4). Ada is the integration of a language for programming in the small with a language for programming in the large. As such it fosters a transparent mapping of both the requirements solution into the design and of that design into the implementation.

Recent research in the software engineering field has produced a number of techniques or rationals for structuring the understanding of systems. The Ada language was designed to facilitate the representing of designs produced using these techniques. Thus Ada as a system design language (SDL) fosters the use of these techniques.

The rest of this chapter will present three methods of structuring systems and show how Ada documents the resulting designs. The three methods are functional decomposition(3), information hiding(18) and abstract data types(7). These techniques are not used in exclusion of one another but rather are complementary of each other. They are normally used in conjunction with one another for the complementary tasks of refining and enhancing the design during the system's development process.

1. Functional decomposition.

The functional decomposition or data transformation method of design will be illustrated as it appears in the analysis and design technique of structured analysis. Structured analysis(3) is a methodology which uses a graphical method for functionaly specifying the structure of the system being designed. This method uses graphs known as "data flow diagrams" in conjunction with logical data descriptions stored in data dictionaries to model the structure of the system. The data flow diagram documents the structure of the system as the logical flow of data items through the system, shown as labeled arrows, and the transformations which happen to these data flows, shown as labeled circles or boxes. The structure of a data item is documented by a logical definition, including the naming and defining of its components, in the data dictionary.

In use the logical data flows and data transformations are derived in a three step process. First an existing or envisioned physical system is modelled, resulting in a data flow diagram which is labeled with both physical data items and physical transformations. In the second phase the physical model is transformied into a logical model by abstracting logical data flows and transforms. Finally the resulting logical model is modified until it models a system which fulfills the requirements.

The results of this process can be seen in the following example design of a stoplight control system: The transformations of data types (data flows) which take place in a stoplight system are diagrammed in figure 2. The presence or absence of vehicle in a lane as observed by a detector is transformed into an approach being either occupied or empty.

The state of the approach is transformed into a request for a green or red light. This request causes the light to be set to red, yellow or green as appropriate considering the current state of the system. Note that this method of decomposition seeks first to discover, and name, the logical types of data which exist in the system and then name the activities which transform these types.

When these data types (arrows) and their transformations (boxes) are named and connected, the "data flow diagram" shown in FIG 2 results. This diagram can then be used as a basis to model the structure of the system in Ada. This method of design produces modules (transformations) which are defined by the interfaces (data flows) between the modules. This structure is easily transformed into Ada where the modules become Packages or Tasks and the interfaces become the visible parts of these modules (ie. types, object, procedures, functions or entries).

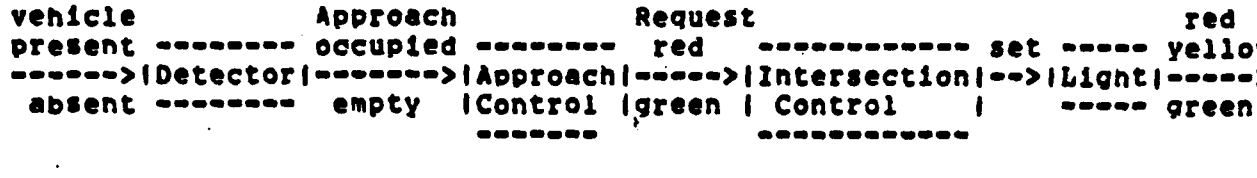


FIG 2.

This data flow diagram is transformed into the following design which is documented as the visible part of an Ada program(FIG 3). In the design the modules detector, approach control and intersection control have two tasks each, one for each direction. The complete system is included in the appendix.

```
procedure STOP_LIGHT IS  -- main program.
    type DIRECTION is (N_S,E_W);
  package DETECTOR is
      task type CONTROL_TASK ;  -- is hardware.
    CONTROL:array(DIRECTION) of CONTROL_TASK;
    end DETECTOR;
  package APPROACH is
      task type CONTROL_TASK is
        entry APPROACH_OCCUPIED; -- interrupt.
        entry APPROACH_EMPTY;    -- interrupt.
        end CONTROL_TASK;
    CONTROL:array(DIRECTION) of CONTROL_TASK;
    end APPROACH;
  package INTERSECTION is
      task type CONTROL_TASK is
        entry REQUEST_GREEN;
        entry REQUEST_RED;
        end CONTROL_TASK;
    CONTROL:array(DIRECTION) of CONTROL_TASK;
    end INTERSECTION;
  package LIGHT is
    task CONTROL is
      entry SET_TO(COLOR)(DIR: in DIRECTION);
      end CONTROL;
    end LIGHT;
end STOP_LIGHT;
```

FIG 3.

The transformation from data flow diagram to program structure (visible part of Ada program) was straightforward in this case. The modules became packages and the interfaces became entries or procedures. These entries or procedures become properties, or resources, of one of the modules which it connects.

Structured analysis is one of a number of functional decomposition methodologies which derive the structure of a system in terms of the transformations or functions on logical data types. The transformations produce the logical data types which occur in a system from the system's other logical data types. Other functional decomposition methodologies which produce similar system designs are SADT(21), HOS(17) and data directed decomposition(16).

2. Information hiding.

Information hiding(18) is a method of modularization in which each module hides the irrelevent information about components inside the module while providing access to the required information to enable use of the module. The interface of a module provides an abstract view(19) of the entity enclosed within it. The entities which are enclosed in the modules are the design decisions which must be made during the design process.

To see why it is that the design decisions are the entities in the modules, the rational for this methodology must be described. One of the major failings in the normal methods of designing systems is that they produce systems which are expensive to maintain. The reason that this is true is that the systems are difficult to change. Since changing a system means changing some design decision, it follows that if one wants a system to be easy to change (maintain) then each design decision must affect as little of the system as possible and there should be as little coupling between design decisions as possible. This last sentence was the rational which brought the information hiding design method into existence. Restated it says in order for a system to be maintainable, it should be designed as an interconnection of modules where each module hides the result of one design decision by presenting for use by the other modules an abstract view of the entity about which that decision was made.

When a system is being designed using this methodology, certain guidelines are used when making the design decisions which determine the decomposition of the system into components. These components can then be independently designed, and after the system is in use, independently modified. Essentially, the information hiding guidelines are the following(18):

1. Make a list of all of the design decisions for which change cannot be ruled out.
2. Encapsulate each design decision in one module. This means that all of the programs that require the knowlege of the decision, and only those, comprise the module.
3. Design the abstract interface. This interface consists of the data types and procedures that the module users need to be able to make use of the entity in the module without knowing how it is implemented.

As an example of this methodology, in the design of a text editor, the way in which the text is stored in the editor is a design decision. For instance the data may be stored in memory,or it may be stored on a disk file, or it may be stored in virtual memory. Also it may be stored as an array or a list of some kind. Therefor the entity to be designed is made the contents of a module. This module called DOCUMENT_HOLDER is shown in FIG 4 as an Ada package specification (ie.visible part).

```
package DOCUMENT_HOLDER is
    type LINE is STRING(LINE_LENGTH);
    type LINE_NUMBER is private;
  procedure CLEAR;
  function EMPTY return BOOLEAN;
  function NEXT_LINE return LINE_NUMBER;
  function PREVIOUS_LINE return LINE_NUMBER;
  function FIRST_LINE return LINE_NUMBER;
  function LAST_LINE return LINE_NUMBER;
  procedure INSERT_BEFORE (LINE_NUM:in LINE_NUMBER; THIS_LINE:in LINE);
  procedure INSERT_AFTER (LINE_NUM:in LINE_NUMBER; THIS_LINE:in LINE);
  procedure GET_LINE  (LINE_NUM:in LINE_NUMBER; THIS_LINE:out LINE);
end DOCUMENT_HOLDER;
```

FIG 4.

As can be seen the Ada package specification is just the abstract interface needed for this methodology. This package specifies that DOCUMENT_HOLDER is a collection of numbered lines which can be operated on by using the procedures and functions and can be implemented in any way as long as the implementation provides the specified types, procedures and functions.

3. Abstract data types.

Data abstraction is a "thought tool" as well as a methodology. The way in which the designer of a system approaches the design is greatly influenced by the language in which he expresses designs. The availability of data abstraction facilities in a system design language like Ada provides the designer with the ability to modularize the system into the logical entities (abstract data objects) most appropriate to the problem beign solved.

The use of abstraction in the design of systems is one of the main ways of reducing the complexity in the system's design. The reduction of complexity is accomplished by concentrating on defining only the essential logical characteristics of the system and ignoring for the time being, the nonessential implementation details of the system. This concentration upon logical properties leads to the specification of the system as an abstract data type, or object, consisting of the type name and the operations which are associated with the type. Thus the first step in the design process results in the system being specified as an abstract data type in Ada, that is a package specification consisting of a type declaration and a set of operations (procedures or functions) applicable to objects of that type.

Once the system is specified, and thought of, as an abstract data type(19), the types and operations needed to logicaly implement the system are conceived and specified as abstract data types. This is accomplished by a process of successively refining(16) the data abstractions in terms of other more concrete data types. This successive refinement process is an iterative process whereby the abstract data types which are needed by the program at one level are implemented, or represented, as data types which are less abstract or more concrete which are then implemented in terms of even less abstract data types, etc. This decomposition of abstract data types into less abstract types continues until the data types needed are available directly in the programming language being used to implement the system.

As an example, a one pass assembler may be described at the most abstract level as consisting of an assembly procedure which gets symbols from a source file and using a symbol table for storage puts data and instructions in an object file. The structure of the assembler is shown in FIG 5. Each of the named boxes is an abstract data type.

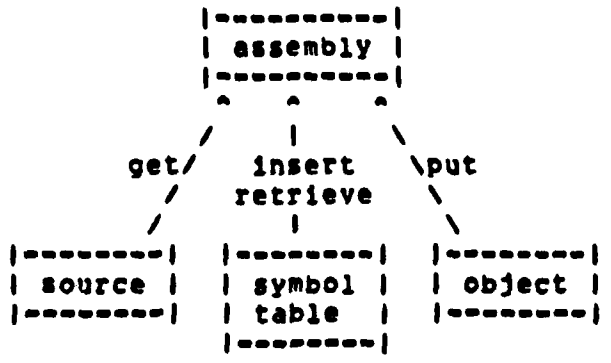


FIG 5

The specification of the abstract data type symbol table as an Ada package is shown in FIG 6a. Note particularly that the symbol table is not the set of memory locations and storage patterns that is classicly thought of as describing a table, but rather it is a set of logical operations applicable to the type symbol table. A significant additional advantage of decomposing systems into abstract data types is that some of the data types may be generally useful in other systems. Since the interface to an abstract data type is simple, by definition its interface is the minimum necessary information needed to use it, the abstract data type is the ideal entity to build libraries around. An example of a generally useful data type is the character buffer shown in FIG 6b.

```
package SYMBOL_TABLE is
  type SYM_NAME is new STRING(20);
  type SYM_TYPE is (INT,FLOAT)
  type ITEM is
    record
      NAME:SYM_NAME;
      TYPE:SYM_TYPE;
      LOCATION:INTEGER;
    end record;
  procedure INSERT(SYMBOL:in ITEM);
  procedure RETRIEVE(NAME:in SYM_NAME;
                     SYMBOL:out ITEM);
  NOT_FOUND:exception;
            --raised by retrieve
  end SYMBOL_TABLE;
```

FIG 6a

```
task BUFFER is
  entry READ(C:out CHARACTER);
  entry WRITE(C:in CHARACTER);
end BUFFER;
```

FIG 6b

4. Documentation.

A significant advantage of using Ada as the means of documenting the structure of the system is that it is maintainable using the same methods and automated tools used to maintain computer programs. In the case where Ada is also the implementation language, the maintenance of the design documentation becomes automatic since it is an integral part of the implementation

The actual documentation of the structure of the system is contained in the visible parts of the packages which constitute the specification of the system. Since these package specifications are textually nested as required by the language processor and formally provide only the syntax of the abstract data types, it is wise to supplement the Ada specification in two ways. First, since people understand the structure of things better when presented with a graphical representation, the Ada text should be supplemented with its equivalent graphical rendition in a form similar to FIG 1a.

Secondly, since the Ada text provides only the syntax of the provided types and operations, that text should be supplemented with both descriptive semantic information and a list of the required types and operations, included as comments.

The system documentation should also include structured requirements specification(12) in a form compatible with the design specification. In addition, the development of the requirements should benefit from the use of a structured analysis technique such as HOS (3,17). A complete system description of the stoplight control system shown in FIGs 2 and 3 is included in appendix I as as example of the documentation of a system . Appendix II provides a syntax description of Ada as a System Design Language (SDL) and describes its usage.

V. System Component Design.

The design process for the three types of program units is basicly similar. This process is based on the small scale structuring mechanisms of the Ada language: strong data typing, user definable data types, structured control constructs and procedure or function invocation. Ada provides a strong typing mechanism whereby every object must have a declared type and conversion between objects of different types is not allowed. During compilation, this mechanism catches a number of design errors which occur frequently in nontyped languages.

Types in Ada consist of the built in numeric types, enumeration types, where the programmer defines by name the values of the type, and certain built in enumeration types such as boolean and character types. Additionaly, Ada provides user defined composite types whereby the programmer can define arrays with elements of the same type and records with elements of different types. Ada also provides the ability to create dynamic objects of any type by defining access types to those objects.

The control constructs consist of sequencing of statements, an alternative choice mechanism, an iteration construct and procedure or function invocation. The choice mechanism consists of three statements, the "if then else" statements, the "case" statement and, in the case of tasks, the "select" statement. The iteration mechanism is the "loop" statement with a number of termination methods. The basic loop is nonterminating, however it can be modified by either a "for" statement to loop a certain number of times, or a "while" statement to test for a certain condition before each loop. The procedure invocation occurs as a statement and the function invocation occurs in an expression thereby naming needed operations which are defined elsewhere.

At the logical level, a procedure or function defines a single abstract event. A package defines an abstract data type which provides operations on permanent objects of the type when requested. A task defines an abstract data type but it is active and operates in parallel with other tasks by either providing or asking for operations with those other tasks.

A procedure implementation, for instance FIG 7, may declare some local types, objects, procedures, functions, packages or tasks. These declared entities are only visible within the procedure and have a lifetime limited to the current invocation of the procedure. Procedures provide only one operation or event which can be invoked.

```
procedure SORT (A:in INTEGER_ARRAY) is -- bubble sort
   procedure EXCHANGE (LEFT,RIGHT:in out INTEGER) is
      TEMP:INTEGER;
      begin
         TEMP:=LEFT;
         LEFT:=RIGHT;
         RIGHT:=TEMP;
      end EXCHANGE;
   begin
      for LAST in reverse A'RANGE loop
         for FIRST in A'FIRST..LAST loop
            if A(FIRST) > A(FIRST+1) then
               EXCHANGE (A(FIRST),A(FIRST+1));
               end if;
            end loop;
         end loop;
   end SORT;
```

FIG 7

A package implementation, for instance FIG 8, may also declare some local types, objects, procedures, functions, packages or tasks. These declared entities are, again, only visible within the package but they now have a lifetime which is not limited to the invocation of one of the procedures in the package's visible part, but rather last as long as the package lasts. Package specifications provide all of the operations which are allowed to be applied to the enclosed entities.

```
package body SYMBOL_TABLE is
   subtype INDEX is INTEGER range 1..200;
   TABLE: array(INDEX) of ITEM;
   function FIRST_FREE return INDEX is ... end;
   procedure INSERT(SYMBOL:in ITEM) is
      begin
         TABLE(FIRST_FREE):=SYMBOL;
      end INSERT;
   procedure RETRIEVE(NAME:in SYM_NAME;SYMBOL:out ITEM) is
      begin
         for I in INDEX loop
            if NAME=TABLE(I).NAME then
               SYMBOL:=TABLE(I);
               return;
               end if;
         end loop;
         raise NOT_FOUND;
      end RETRIEVE;
begin
  -- initialize table
end SYMBOL_TABLE;
```

FIG 8 (Implementation of FIG 6a)

A task implementation (body) may, like a package, declare local types, objects, procedures, functions, packages and tasks, but the task can only export entries. These local declarations have, as in the case of packages, a lifetime equal to that of the task. The control structures in tasks must however, be supplemented with constructs not available in procedures or packages since the order in which entries may be called and the syncronization with other tasks must be represented. The structure of control in tasks is based on the selection of those entries which may be accepted at the current state in processing the task. In addition the main part of the task body is normally enclosed in an infinite loop, since tasks normally run until explicitly terminated.

An example task body for the task BUFFER of FIG 6b is shown in FIG 9 and a complete system based on tasks is the stoplight system shown in appendix I.

```
task body BUFFER is
   SIZE:constant INTEGER:=10;
   BUFF:array(1..SIZE) of CHARACTER;
   SLOTS_FULL:INTEGER range 0..SIZE:=0;
   WRITE_INDEX,READ_INDEX:INTEGER range 1..SIZE:=1;
begin
   loop
      select
         when SLOTS_FULL < SIZE =>
            accept WRITE(C:in CHARACTER) do
               BUFF(WRITE_INDEX):=C;
               end;
            WRITE_INDEX:= WRITE_INDEX mod SIZE +1;
            SLOTS_FULL:=SLOTS_FULL+1;
         or when SLOTS_FULL > 0 =>
            accept READ(C:out CHARACTER) do
               C:=BUFF(READ_INDEX);
               end;
            READ_INDEX:=READ_INDEX mod SIZE+1;
            SLOTS_FULL:=SLOTS_FULL-1;
         or
            terminate;
         end select;
      end loop;
   end BUFFER;
```

FIG 9 (Implementation of FIG 6b)

The small scale structuring mechanisms in the Ada language are constraints on the structure of the components from which systems are constructed. Their purpose is not to assist in the development of algorithms but rather in forcing the implementation of the algorithm to clearly display the logic of the algorithm in terms appropriate to the algorithm. In addition these mechaninisms cause the information necessary to understand the algorithm, namely the information used by the algorithm, to be highly localized. Thus Ada as a program design language (PDL) enhances the maintainability of the components of a system in an analogous way to the way Ada as a System Design Language enhances the maintainability of the system. That is, Ada enhances maintainability by constraining structure so that the structure of the system (or program) clearly displays the logic of the solution to the problem and the information necessary to understand any part of the system is local to that part.

VI. Conclusions.

The use of Ada as a System Design Language (SDL) imposes some significant constraints on the modularizations which can be used in the design of systems. The imposition of these constraints is the source of the strength of Ada as the SDL in the same way that the imposition of constraint on the control structure of programs is the source of strength of structured programming. Good systems like good programs have a structure that is discernable to the reader and have a good localization of names and concepts in the component parts. Thus the structured programming and the modularization required by Ada reduces the complexity(17) of system designs and implementations.

The constraints which Ada imposes on designs are in how one may specify the components from which one can construct the system and how these components can be interconnected to form the system. The specification of a component in Ada is the specification or visible part of an Ada package. This is exactly the abstract interface (19) syntax required by the modern modularization methods. A document consisting of the Ada package specifications which constitute a system, ie. an SDL description, supplemented with its graphical equivalent is an ideal formal design description from the points of view of understandability and maintainability(15). When this design description is implemented in the Ada language, an additional twofold advantage is accrued. Both the design and the implementation mirror the structure of the problem and the design documentation is automatically maintained since it is an integral part of the finished system.

REFERENCES.

1. Ada reference manual, July 1980.

2. Caine, S. & Gordon, E. PDL-A tool for software design. Proceedings, National Computer Conference,1975.

3. Demarco, T. Structured analysis and system specification Prentis Hall 1979.

4. DeRemer,F. & Kron,H. Programming in the large versus programming in the small. IEEE trans S.E. June,1976.

5. Design and implementation of programming languages. Springer-Verlag:Lecture notes in computer science (54).

6. Dijkstra, E.W. The structure of the "THE" multiprogramming system. C. ACM 11 (May 1968),341-346.

7. Guttag,J. Abstract data types and the development of data structures. C.ACM 20,6,(June 1977),396-404.

8. Guttag,J. Notes on type abstraction. Proceedings,Specification of reliable software IEEE 1979. & IEEE Trans S.E. Jan 1980.

9. Habermann,A.N. ,Flon,L. & Cooprider, L. Modularization and hierarchy in a family of operating systems. C.ACM 19,5,(May 1976),266-272.

10. Habermann,A.N. & Perry,D. Well formed system compositions. Carnegie Mellon Univ. Comp Sci Rept Mar 1980.

11. Higher Order Software Tech Rept no.4 AXES syntax description Dec 1976.

12. Heninger,K. Specifying software requirements for complex systems: New techniques and their application. Proceedings,Specification of reliable software IEEE 1979. & IEEE Trans S.E. Jan 1980.

13. Jackson,K. Parallel processing and modular software construction. Design and implementation of programming languages. Springer-Verlag:Lecture notes in computer science (54).

14. Jensen & Wirth Pascal users manual. Springer-Verlag.

15. Jones,C. A survey of programming design and specification techniques. Proceedings,Specification of reliable software IEEE 1979.

16. Morris, J.B. Programming by successive refinement of data abstractions. Software practice and experience, Apr 1980.

17. Pager, D. On the problem of communicating complex information. C.ACM 16,5,(May 1973),275-281.

18. Parnas, D. A technique for software module specification with examples. C.ACM 15,5,(May 1972),330-336.

19. Parnas, D. Use of abstract interfaces in the development of software for embedded systems. NRL Report 8047,1977.

20. Rational for the design of the Ada programming language. ACM Sigplan notices 14,6,(June 1979), Part B.

21. Ross,D.T. Structured analysis(SA):A language for communicating ideas. IEEE Trans S.E. Jan 1977.

22. Steelman. DoD HOL Requirements. June 1977.

23. Wirth,N. Program development by stepwise refinement. C.ACM 14,4,(April 1971),221-227.

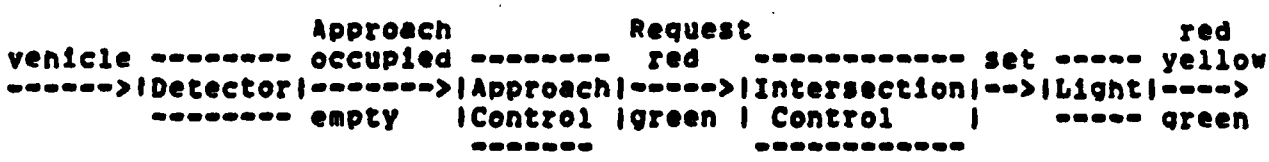
24. Yourdon,E. Structured walkthroughs. Prentis-Hall.

APPENDIX I. DESIGN EXAMPLE: A STOPLIGHT CONTROL SYSTEM.

STOPLIGHT DATA FLOW DIAGRAM.

The transformations of data types which take place in a stoplight are diagrammed in the figure. The detection or non detection of a vehicle in a lane is transformed into that approach being either occupied or empty. The state of the approach is transformed into a request for a green or red light. This request causes the light to be set to red, yellow or green.

When these data types and their transformations are named and connected, the following "data flow diagram" results. This diagram can then be used to model the structure if the system. This method of design produces modules (transformations) whose descriptions are the interfaces between the modules. This structure is easily modeled in Ada where the modules become Packages or Tasks and the interfaces become the visible parts of these modules (ie. types, object, procedures or entries).



STOPLIGHT REQUIREMENTS SPECIFICATION.

1. INTRODUCTION.

This requirements specification describes a stoplight system and its components. The description uses module descriptions to describe the system itself and each of the hardware and software components. The module descriptions give only the external behavior (interface characteristics) of the system or component. These interface characteristics are just the visible behavior of the module and are described independent of other modules which this module may be connected to. This means that, for instance, hardware modules descriptions refrain from desribing the system effects which the module causes or displays. Software modules likewise refrain from describing their effect on either the hardware or other software modules but merely the functions that it provides.

2. STOPLIGHT SYSTEM.

MODULE NAME: STOP_LIGHT

BEHAVIOR: The stoplight system controls a signal at a four way intersection. It detects vehicles in an approach area of each approaching lane of the intersection and provides a green light to occupied lanes.

When no lanes are occupied all lights are red.

When a lane becomes occupied the light in its direction is made green.

As long as the lane remains occupied the light in its direction remains green. If however, a lane in the opposite direction becomes occupied then, after 15 seconds, the light in the current direction goes through yellow to red and the light in the opposite direction becomes green.

When a lanes in a direction become empty the light in that direction goes through yellow to red.

3. HARDWARE MODULES.

MODULE NAME: DETECTOR

BEHAVIOR: Detects when a lane approach becomes occupied and signals that its lane is occupied (ie. generates an occupied interrupt). Detects when a lane becomes empty and signals that its lane is empty (ie. generates an empty interrupt).

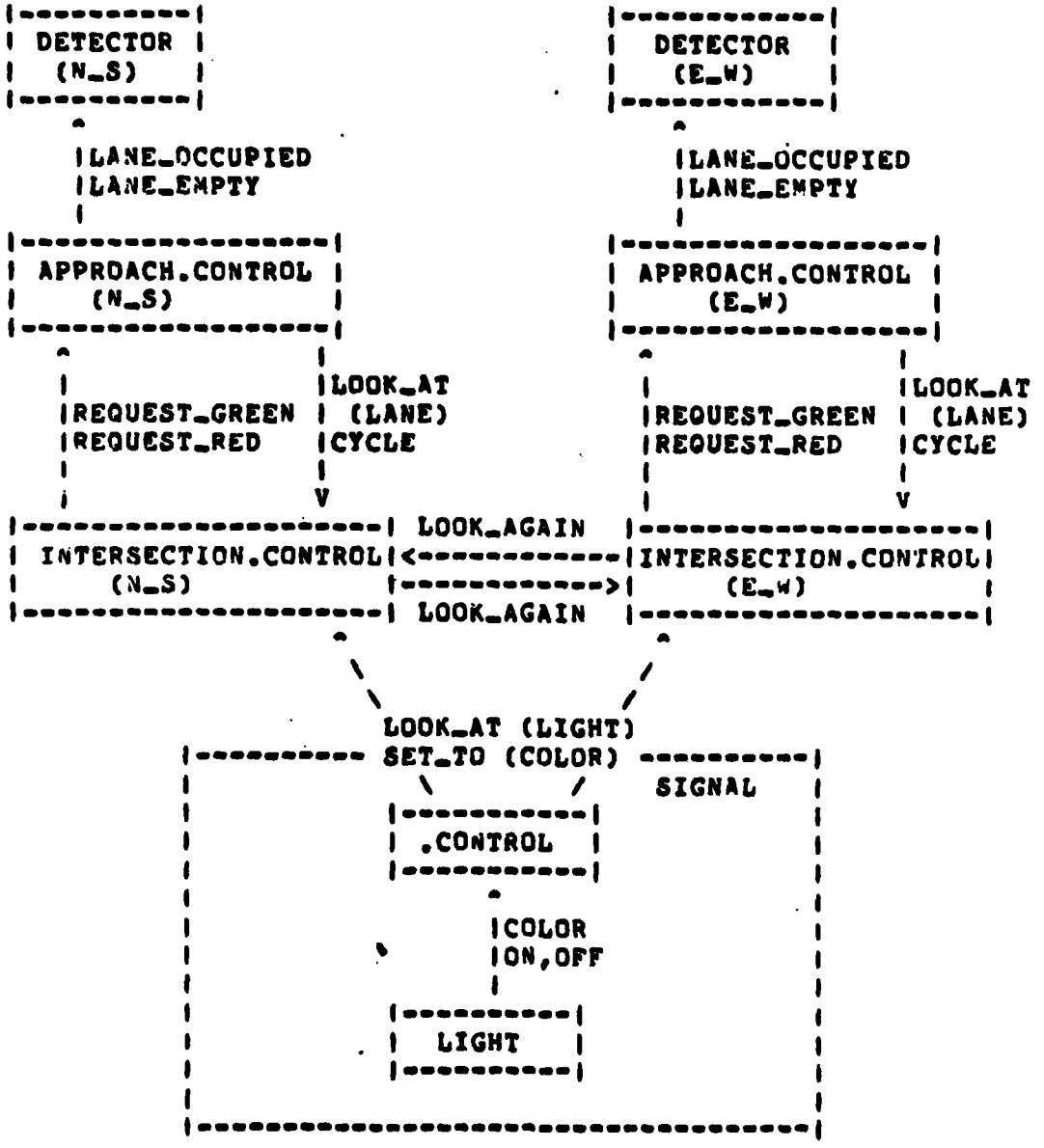
There are two detectors, one which responds to lanes in the north-south direction (the north detector is ORed with the south detector), and one of which responds to the east-west direction.

CHARACTERISTIC VALUES:

N-S	OCCUPIED	=> INTERRUPT AT LOCATION 8
N-S	EMPTY	=> INTERRUPT AT LOCATION 16
E-W	OCCUPIED	=> INTERRUPT AT LOCATION 12
E-W	EMPTY	=> INTERRUPT AT LOCATION 20

```
-----------------------------------------------------------------
     MODULE NAME: LIGHT

     PROVIDES: Type COLOR is (RED,YELLOW,GREEN)
               LIGHT(N_S,E_W)

     BEHAVIOR: LIGHT = array(N_S,E_W) of three lamps with red,yellow and
                       green filters. Each lamp can be on or off.

     CHARACTERISTIC VALUES:
                LIGHT  = LOCATIONS 76,80
                RED    = b 100
                YELLOW = b 010
                GREEN  = b 001
-----------------------------------------------------------------

4. SOFTWARE MODULES.
-----------------------------------------------------------------
     MODULE NAME: APPROACH.CONTROL

     PROVIDES: LANE_OCCUPIED
               LANE_EMPTY
               LOOK_AT(LANE)
               CYCLE

     REQUIRES: REQUEST_RED
               REQUEST_GREEN

     BEHAVIOR: Model state (empty, occupied) of lane.
               LANE_OCCUPIED causes occupied & REQUEST_GREEN.
               LANE_EMPTY causes empty & REQUEST_RED.
               LOOK_AT shows state.
               CYCLE behaves like EMPTY followed by OCCUPIED.
-----------------------------------------------------------------
     MODULE NAME: INTERSECTION.CONTROL

     PROVIDES: REQUEST_RED
               REQUEST_GREEN

     REQUIRES: LOOK_AT(LIGHT),LOOK_AT(LANE),SET_TO(COLOR)(DIR),CYCLE

     BEHAVIOR: When lights are red REQUEST_GREEN causes
                SET_TO(GREEN)(MY_DIRECTION).
               When its light is green REQUEST_RED causes
                SET_TO(YELLOW)(MY_DIRECTION) then SET_TO(RED)(MY_DIRECTION).
               When its light is green AND its approach is continuously
                occupied AND, after 15 seconds, the other approach becomes
                occupied cause CYCLE.
-----------------------------------------------------------------
     MODULE NAME: SIGNAL.CONTROL

     PROVIDES: SET_TO(COLOR)(DIRECTION)
               LOOK_AT(LIGHT)

     REQUIRES: LIGHT(DIR)(COLOR(ON,OFF))

     BEHAVIOR: Model state (N-S,E-W)(RED,YELLOW,GREEN) of light.
               SET_TO causes LIGHT(DIRECTION):= COLOR.
               LOOK_AT copies the state of LIGHT.
-----------------------------------------------------------------
```

STOPLIGHT CONTROL SYSTEM DIAGRAM



```
procedure STOP_LIGHT_CONTROL_SYSTEM   is -- Main program
                                                -- System Specification
  type DIRECTION is (N_S,E_W);                  -- SDL

package DETECTOR is
    task type DETECTOR_TASK ;
  DETECTOR:array(DIRECTION)of DETECTOR_TASK;
      -- Detects vehicles in the approach to the intersection and
      -- signals the approach controller in its direction when the
      -- approach becomes occupied or empty.
  --REQUIRES: LANE_OCCUPIED, LANE_EMPTY
  end DETECTOR;

package APPROACH is
    type LANE is (EMPTY,OCCUPIED);
    task type CONTROL_TASK is
      entry LANE_OCCUPIED; --interrupt.
      entry LANE_EMPTY;    --interrupt.
      entry LOOK_AT(L:out LANE);
      entry CYCLE; --Behaves like EMPTY followed by OCCUPIED.
        -- OCCUPIED causes a request for a green light.
        -- EMPTY causes a request for a red light.
      entry INITIALIZE_TO(DIR:in DIRECTION);
      end CONTROL_TASK;
  CONTROL:array(DIRECTION)of CONTROL_TASK;
      for CONTROL(DIRECTION'FIRST).LANE_OCCUPIED use at 8;
      for CONTROL(DIRECTION'LAST).LANE_OCCUPIED use at 12;
      for CONTROL(DIRECTION'FIRST).LANE_EMPTY use at 16;
      for CONTROL(DIRECTION'LAST).LANE_EMPTY use at 20;
  --REQUIRES: REQUEST_GREEN; REQUEST_RED;
  end APPROACH;

package INTERSECTION is
    task type CONTROL_TASK is
      entry REQUEST_RED;
      entry REQUEST_GREEN;
      entry LOOK_AGAIN; -- look at the light and the intersection again.
      entry INITIALIZE_TO(DIR:in DIRECTION);
      end CONTROL_TASK;
  CONTROL:array(DIRECTION)of CONTROL_TASK;
      -- Provides green lights to occupied roads and red lights to
      -- empty roads while alternating when necessary.
  --REQUIRES: LOOK_AT(LIGHT); LOOK_AT(LANE); SET_TO(COLOR)(DIRECTION);
  --          CYCLE; OTHER_DIRECTION.LOOK_AGAIN;
  end INTERSECTION;

package SIGNAL is
  type COLOR is (RED,YELLOW,GREEN);
    for COLOR use (RED=>2#100#,YELLOW=>2#010#,GREEN=>2#001#);
  type STOP_LIGHT is array(DIRECTION) of COLOR;
  LIGHT:STOP_LIGHT:=(RED,RED);
    for LIGHT use at 76;
  task CONTROL is
    entry SET_TO(COLOR)(DIR:in DIRECTION);
    entry LOOK_AT(L:out STOP_LIGHT);
      -- SETs light in one direction to red,yellow or green
      -- while setting other direction to red.
    end CONTROL;
  --Includes the module LIGHT from which it
  --REQUIRES:LIGHT(DIRECTION,COLOR(ON,OFF));
  end SIGNAL;                                   -- End Specification
```

```
package body DETECTOR is                          --Begin System Implementation
                -- Hardware.
end DETECTOR;


package body APPROACH is
  task body CONTROL_TASK is
    MY_LANE:LANE :=EMPTY;
    MY_DIR:DIRECTION;

  begin -- APPROACH.CONTROL_TASK
    accept INITIALIZE_TO(MY_DIR:in DIRECTION); --learn my direction
    loop
      select
        accept LOOK_AT(L:out LANE) do
          L:=MY_LANE;
          end;
        or accept LANE_OCCUPIED do
          MY_LANE:=OCCUPIED;
          INTERSECTION.CONTROL(MY_DIR).REQUEST_GREEN;
          end;
        or accept LANE_EMPTY do
          MY_LANE:=EMPTY;
          INTERSECTION.CONTROL(MY_DIR).REQUEST_RED;
          end;
        or accept CYCLE; --simulate break in steady stream of traffic.
          if MY_LANE=OCCUPIED THEN  -- if necessary.
            INTERSECTION.CONTROL(MY_DIR).REQUEST_RED;
            INTERSECTION.CONTROL(MY_DIR).REQUEST_GREEN;
          end if;
      end select;
    end loop;
  end CONTROL_TASK;
end APPROACH;


package body INTERSECTION is
  task body CONTROL_TASK is
    MY_DIR:DIRECTION;
    OTHER_DIR:DIRECTION;
    OTHER_LANE:APPROACH.LANE;
    LIGHT:SIGNAL.STOPLIGHT;

  begin  -- INTERSECTION.CONTROL_TASK
    accept INITIALIZE_TO(MY_DIR:in DIRECTION) do --learn my direction
      if MY_DIR=DIRECTION'LAST then
        OTHER_DIR:=DIRECTION'FIRST;
      else
        OTHER_DIR:=DIRECTION'SUCC(MY_DIR);
      end if;
    end INITIALIZE_TO;
    loop
      SIGNAL.CONTROL.LOOK_AT(LIGHT);
      select
        when LIGHT=(RED,RED) =>
          accept REQUEST_GREEN do
            SIGNAL.CONTROL.SET_TO(GREEN)(MY_DIR);
            INTERSECTION.CONTROL(OTHER_DIR).LOOK_AGAIN;
          end;
        or when LIGHT(MY_DIR)=RED => -- catch extra red requests
          accept REQUEST_RED;
```

```
        or when LIGHT(MY_DIR)=GREEN =>
          accept REQUEST_RED do
            SIGNAL.CONTROL.SET_TO(YELLOW)(MY_DIR);
            delay 3*SECONDS;
            SIGNAL.CONTROL.SET_TO(RED)(MY_DIR);
            INTERSECTION.CONTROL(OTHER_DIR).LOOK_AGAIN;
          end;
        or when LIGHT(MY_DIR)=GREEN =>
          delay 15*SECONDS;
          APPROACH.CONTROL(OTHER_DIR).LOOK_AT(OTHER_LANE);
          if OTHER_LANE=OCCUPIED then
            APPROACH.CONTROL(MY_DIR).CYCLE;
            end if;
        or accept LOOK_AGAIN;
      end select;
    end loop;
  end CONTROL_TASK;
end INTERSECTION;

package SIGNAL is
  task body CONTROL is

  begin  -- SIGNAL.CONTROL
    loop
      select
        accept LOOK_AT(L:out LIGHT) do
          L:=LIGHT;
          end;
        or when LIGHT=(RED,RED) =>
            accept SET_TO(GREEN)(DIR: in DIRECTION) do
            LIGHT(DIR):=GREEN;
          end;
        or accept SET_TO(YELLOW)(DIR:in DIRECTION) do
            LIGHT(DIR):=YELLOW;
          end;
        or accept SET_TO(RED)(DIR:in DIRECTION) do
            LIGHT(DIR):=RED;
          end;
      end select;
    end loop;
  end CONTROL;
end SIGNAL;


begin -- STOP_LIGHT_CONTROL_SYSTEM  Main program
  for DIR in DIRECTION loop --make controllers aware of their directions
    APPROACH.CONTROL(DIR).INITIALIZE_TO(DIR);
    INTERSECTION.CONTROL(DIR).INITIALIZE_TO(DIR);
  end loop;
end STOP_LIGHT_CONTROL_SYSTEM;                    -- End System Implementation
```

APPENDIX II. SYSTEM DESIGN LANGUAGE
SYNTAX INFORMATION

SYNTAX SUMMARY

```
compilation ::= {compilation_unit}
compilation_unit ::= {WITH unit_name{,unit_name}}subprogram_declaration;
                   | {WITH unit_name{,unit_name}}subprogram_body;
                   | {WITH unit_name{,unit_name}}package_declaration;
                   | {WITH unit_name{,unit_name}}package_body;
                   | {WITH unit_name{,unit_name}}subunit;
subprogram_declaration ::= PROCEDURE identifier[formal_part]
                         | FUNCTION identifier[formal_part]RETURN type
subprogram_body ::= subprogram_declaration IS
                      {declarative_item}
                    BEGIN
                      {statement}
                    END
package_declaration ::= PACKAGE identifier IS
                          {declaritive_item}
                          [PRIVATE
                          {declaritive_item}]
                        END
package_body ::= PACKAGE BODY identifier IS
                   {declaritive_item}
                 [BEGIN
                   {statement}]
                 END
subunit ::= SEPARATE(unit_name) subunit_body
declerative_item ::= object_declaration
                   | type_declaration
                   | subprogram_declaration
                   | package_declaration
                   | task_declaration
```

RATIONAL AND USAGE

In Ada, system structure has two views: The textual system structure and the physical system structure. The textual system structure is the textual layout of the program and is portrayed by the systematic nesting of program units(packages, subprograms and tasks) within declarative parts of other program units. Also, the specification of a program unit (ie. its benavior definition) is textually separate from the body of that program unit(ie. its implementation). This textual structuring of the system accomplishes the grouping of semantically related units and controls the scope of names in the system.

The physical system structure is the grouping of program units into compilation units. Each program unit specification and body is a compilation unit and is compilable separately from the other compilation units. The visibility, or allowed usage, of items declared in other compilation units must be provided explicitly, using a WITH(other_unit) clause, in a compilation unit. This allows precise control of the names usable in a unit.

The textual structuring mechanism provides the basic control of the visibility of named entities via nesting and separation of behavior from implimentation, while the physical structuring mechanism provides the additional capability of explicit control of the dependencies of units on other units which are textualy visible to the unit.

Textually, a system in Ada is normally presented as a procedure, giving a name to the system, which has a (large) declarative part. The declarative part consists of types and objects global to the system, a sequence of package specifications which define the components of the system and a sequence of package bodies which implement the components. The (small) sequence of statements of the "main" procedure serves to initialize the components and start the operation of the system.

For example, textually a system appears like the following:

```
procedure SYSTEM is
  type GLOBAL_TYPES is
  OBJECT: TYPE;
     ...
  package FIRST is
    type ...
    OBJECTS ...
    procedure specifications
        &
    function specifications
  end FIRST;
  package SECOND is
     ...
  end SECOND;
  package THIRD is
     ...
  end THIRD;

  package body FIRST is
    type ...
    OBJECTS ...
    procedure bodies
        &
    function bodies
  end FIRST;
  package body SECOND is
     ...
  end SECOND;
  package body THIRD is
     ...
  end THIRD;

begin -- sequence_of_statements
  initialize;
  start;
end SYSTEM;
```

Layered on top of this textual structure is the physical structure of compilation units. This provides three advantages not possible with just textual structuring:

Separability of the development of components from one another, thereby allowing the development of components in parallel.

Increased control of the development of the system, since the designer has control of the design (specification parts) while the programmers have control of only their component's implementation.

Increased clarity in the design, since dependencies among modules are made explicit by the use of "with" clauses.

The above example can be divided into compilation units in either of the following two ways: (dashed lines separate compilation units)

COLLECTION OF UNITS

```
package GLOBAL is
  type GLOBAL ..
  OBJECTS:GLOBAL ..
-------------------------
  with (GLOBAL)
package FIRST is
    ...
end FIRST;
-------------------------
  with (GLOBAL,FIRST)
package SECOND is
    ...
end SECOND;
-------------------------
  with (GLOBAL,SECOND)
procedure SYSTEM is
    ...
end SYSTEM;
-------------------------
package body FIRST is
      ...
end FIRST;
-------------------------
package body SECOND is
      ...
end SECOND;
-------------------------
```

UNIT WITH SUBUNITS

```
-------------------------
procedure SYSTEM is
  type GLOBAL ..
  OBJECTS:GLOBAL ..
package FIRST is
      ...
end FIRST;
package body FIRST is separate;
package SECOND is
      ...
end SECOND;
package body SECOND is separate;
begin
      ...
end SYSTEM;
-------------------------
separate (SYSTEM)
package body FIRST is
      ...
end FIRST;
-------------------------
separate (SYSTEM)
package body SECOND is
      ...
end SECOND;
-------------------------
```

Both of these have the same textual nesting but the physical layout of the first (collection of units) is more flexible and better controls the dependencies of components (packages) upon one another, whereas the second (units with subunits) provides an exposition of the textual structure of the system in a single package (ie. SYSTEM). Large systems will probably benefit more from the first style due to its tighter control of structure and greater potential for parallelism in development whereas small systems may benefit from the ease of managing the design within one unit provided by the second style.

Tasks are program units but they are not compilation units. Since most embedded computer systems will need to use tasking, the ability to provide tasks as compilation units can be accomplished as follows:

The task type, its needed type definitions and the task object(s) are declared in the specification part of a package whose sole purpose is to encapsulate the task. The package is then viewed as the component and the body of the task can be separately compilable by making it the subunit of the package specification. For example:

```
-----------------------------------------
     package PACKAGE_NAME is
       types ...

       task type TASK_TYPE_NAME is
          entry FIRST;
             ...
       end TASK_TYPE_NAME;
       task body TASK_TYPE_NAME is separate;

       TASK_OBJECTS:TASK_TYPE_NAME;
     end PACKAGE_NAME;
-----------------------------------------
     separate (PACKAGE_NAME)
     task body TASK_TYPE_NAME is
             ...
     end TASK_TYPE_NAME;
-----------------------------------------
```

END

FILMED

DTIC